AutoReq: Expressing and verifying requirements for control systems

Alexandr Naumchev[*,a,b], Bertrand Meyer[a,c,b], Manuel Mazzara[a], Florian Galinier[b], Jean-Michel Bruel[b], Sophie Ebersold[b]

[a] Innopolis University, Innopolis, Russia
[b] Toulouse University, Toulouse, France
[c] Politecnico di Milano, Milan, Italy

Keywords:
AutoReq
Seamless requirements
Design by contract
AutoProof
Eiffel
Landing gear system
Specification drivers
Multirequirements

ABSTRACT

The considerable effort of writing requirements is only worthwhile if the result meets two conditions: the requirements reflect stakeholders' needs, and the implementation satisfies them. In usual approaches, the use of different notations for requirements (often natural language) and implementations (a programming language) makes both conditions elusive. AutoReq, presented in this article, takes a different approach to both the writing of requirements and their verification. Applying the approach to a well-documented example, a landing gear system, allowed for a mechanical proof of consistency and uncovered an error in a published discussion of the problem.

1. Overview and main results

A key determinant of software quality is the quality of requirements. Inconsistent or incomplete understanding of the requirements can lead to catastrophic results. This article presents a tool-supported method, AutoReq, for producing verified requirements, with applications to control systems. It illustrates it on a standard case study, an airplane Landing Gear System (LGS). The goal is to obtain requirements of high quality:

- Easy to write.
- Clear and explainable to domain experts.
- Amenable to change.
- Supporting traceability through close connections to later development steps, particularly implementation.
- Amenable to mechanical verification and validation.

As the last point indicates, AutoReq includes techniques for not only expressing requirements but also verifying their consistency. The LGS case study illustrated the effectiveness of such verification by uncovering a significant error in a previous description of this often-studied example (Section 6.5).

AutoReq takes natural language requirements and environment assumptions as an input and converts them into a format having the above properties. The new format relies on a programming language with contracts. This viewpoint brings one of the biggest advantages of AutoReq – it makes the requirements verifiable both against the underlying assumptions and future candidate implementations, while maintaining their readability through natural language comments on the code. The present work takes the natural language statements from the LGS case study and translates them to seamless statements, readable and verifiable. The ASM treatment of the case study [6] provides the candidate implementation – an executable ASM specification [21] of the system. This by no means implies applicability of AutoReq to ASMs only. The approach applies to any candidate implementation that follows the small step semantics of ASMs. More precisely, the implementation should run in an infinite loop polling the system environment's state and sending appropriate control signals. To the best of our knowledge, most control systems' implementations follow this approach.

The method of expressing requirements does not introduce any new formalism but instead relies on a standard programming language, Eiffel, using mechanisms of Design by Contract (DbC) [40] to state semantic constraints. While DbC relies on Hoare logic [22], which at first sight does not cover temporal and timing properties essential to the

* Corresponding author.
E-mail addresses: a.naumchev@innopolis.ru (A. Naumchev), b.meyer@innopolis.ru (B. Meyer), m.mazzara@innopolis.ru (M. Mazzara), florian.galinier@irit.fr (F. Galinier), bruel@irit.fr (J.-M. Bruel), ebersold@irit.fr (S . Ebersold).

specification of control systems, we show that it is, in fact, possible and even simple to express such properties in the DbC framework.

The verification part relies on an existing tool, associated with the programming language: AutoProof [55], a program proving framework, which can verify the temporal and timing properties expressed in the DbC framework. Applying it to LGS automatically and unexpectedly uncovered the error. Hoped-for advantages include:

- Expressiveness: requirements benefit both from the expressive power of declarative assertions and from that of imperative instructions.
- Ease of learning: anyone familiar with programming languages has nothing new to learn.
- Continuity with the rest of the development cycle: design and implementation may rely on the same formalism, avoiding the impedance mismatches that arise from the use of different formalisms, and facilitating change.
- Precision: formal specifications (contracts) cover the precise semantics of the system and its environment.
- Existing tools, as available in modern IDEs, that support the requirements process: a compiler for a typed language performs many checks that are as useful for requirements as for code.

The present work, while not claiming to have fully reached these ambitious goals, makes the following contributions:

- The outline of a general method for requirements engineering with application to control systems.
- The use of a programming language as an effective mechanism for requirements specification.
- A precisely defined concept of *verifying requirements* for control systems (complementing the usual concept of verifying programs). This idea originates from [47].
- A translation scheme from temporal and timing properties to simpler Hoare logic properties (essentially, first-order predicates on states) as traditionally used in Design by Contract.
- A simple way to combine *environment* and *machine* aspects (the two components of requirements in the well-known Jackson-Zave approach).
- A direct mapping of these *requirements* concepts into well-known *verification* concepts, *assume* and *assert*.
- The demonstration that it is possible to use an existing *program* prover to verify requirements.

Section 2 discusses consequences of poor requirements. Section 3 presents LGS. Section 4 describes the methodology: how to specify and verify requirements. Section 5 shows how to translate common requirements patterns (originally expressed through temporal logic, timing constraints or Abstract State Machines) into a form suitable for AutoReq. Section 6 sketches the methods application to the case study, including an analysis of the uncovered error. Section 7 discusses related work, and Section 8 discusses limitations and future work.

2. The importance of verifying requirements

Control systems in aerospace, transportation, and other mission-critical areas raise tough reliability demands. Ensuring reliability begins with the quality of requirements: the best implementation is useless if the requirements are inconsistent or do not reflect needs. Requirements for software deserve as much scrutiny as other artifacts such as code, designs, and tests.

The literature contains many examples of software disasters arising from requirements problems of two kinds:

- In the requirements themselves: inconsistencies, incompleteness, inadequate reflection of stakeholders' needs.
- In their relationship to other tasks: design, implementation etc. may wrongly understand, implement or update them.

Examples of the first kind include [32]:

- The year 2000, National Cancer Institute, Panama City: patients undergoing radiation therapy get wrong doses because of a software miscalculation.
- In 1996, Ariane 5 maiden flight fails from flight computer's code crash, out of an uncaught arithmetic exception, in code that was reused from Ariane 4 but relied on assumptions that no longer hold in the new technology.
- In 1990, a bug in software for AT&T's #4ESS long-distance switches crashes computers upon receipt of a specific message sent out by neighbors when recovering from a crash.

Analysis of these examples suggests that the problem lies in part from the use of different methods and of different notations for requirements and other tasks such as implementation. This observation is a basis for the *seamless* approach ([41,42,47,56], following ideas in [52]), which this article applies by using a single notation throughout.

Examples of the second kind include [33]:

- London underground system: several cases [49] of passenger deaths from doors opening or closing unexpectedly, without an alarm notification being sent to the train driver.
- An aerospace project [23] where 49% of requirements errors were due to incorrect facts about the problem world.
- An inadequate assumption about the environment of the flight guidance system, which may have contributed to the crash of a Boeing 757 in Cali [43]. Location information for the pilot to extend the flap arrived late, causing the guidance software to send the plane into a mountain.

These examples and others in the literature illustrate the importance of *verifying* requirements. We will see that it is possible to apply to requirements both the concept of verification, as commonly applied to code, and modern proof-oriented verification tools devised initially for code.

3. The landing gear system

To illustrate AutoReq, this article will use, rather than examples of the authors' own making, the LGS [12], probably the most widely discussed case study in recent control systems literature, e.g. [6,7,11,16,31,36,53].

The Landing Gear System physically consists of the landing set, a gear box that stores the gear in the retracted position, and a door attached to the box (Fig. 1). A digital controller independently actuates the door and the gear. The controller initiates either gear extension or gear retraction depending on the current position of a handle in the

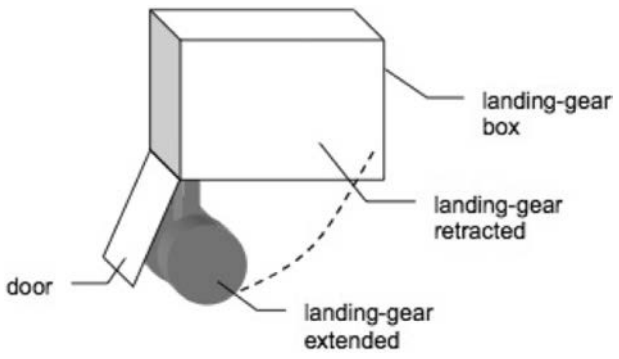

Fig. 1. Landing set (from Boniol et al. [12]).

cockpit. The task is to program the controller so that it sends the correct signals to the door's and the gear's actuators.

The discussion will restrict itself to the system's *normal mode* (there is also a *failure mode*). The defining properties are the following:

$R_{11}bis$: When the landing gear handle has been pushed down and stays down, then eventually the gear will be seen extended and the doors will be seen closed. We interpret this requirement in LTL as $\Box(\Box handle\_down \Rightarrow \diamond(gear\_extended \wedge door\_closed))$ where $\Box$ stands for the *always* temporal operator, and $\diamond$ stands for the *eventually* temporal operator.
$R_{12}bis$: When the landing gear handle has been pulled up and stays up, then eventually the gears will be seen retracted and the doors will be seen closed. We interpret this requirement in LTL as $\Box(\Box handle\_up \Rightarrow \diamond(gear\_up \wedge door\_closed))$.
R_{21}: When the landing gear handle remains in the *down* position, then retraction sequence is not observed. We interpret this requirement in LTL as $\Box(handle\_down \Rightarrow \circ \neg\, gear\_retracting)$ where $\circ$ stands for the *next* temporal operator.
R_{22}: When the landing gear handle remains in the *up* position, then outgoing sequence is not observed. We interpret this requirement as $\Box(handle\_up \Rightarrow \circ \neg\, gear\_extending)$.

We will work not directly from the original description of the LGS but from one of the most interesting treatments of case study [6], which uses the abstract state machine (ASM) approach and applies a process of successive refinements:

1. Start with a *ground model* covering a subset of the requirements.
2. Model-check it.
3. Repeatedly extend (refine) it with more properties of the system, proving the correctness of each refinement.

The AutoReq specification discussed in the next sections starts from the ASM ground model. Some of its features are a consequence of this choice:

- It only accounts for properties specified in the first of the successive models in [6].
- As already noted, it only covers *normal mode*.
- Like the ASM model, it assumes that the only *environment-controlled machine-visible* phenomenon is the pilot's handle [27]. In the failure mode, there might be others.
- It takes over from the ASM model such instructions as $gears := RETRACTED$ which posit that the control system has a way to send the gear directly to the retracted position. This assumption is acceptable at the modeling level but not necessarily true in the actual LGS system.
- The ASM-to-Eiffel translation scheme (Section 5.4) ensures preservation of the one-step semantics of ASM.

4. Requirements methodology

AutoReq builds on the ideas of seamless development [41,56], multirequirements [42] and seamless requirements [47]. The new focus is on requirements verification and reuse of previous requirements through a routine call mechanism. We examine in turn how to specify and reuse requirements and environment assumptions (Section 4.1), and what it means to verify them (Section 4.2).

4.1. Specifying requirements

Specifications in AutoReq, often in practice translated from a document in structured natural language, take the form of contracted Eiffel routines with natural-language comments. These routines are further consumed by:

- The verification tool. Since the routines coming out of the translation process are equipped with contracts, they may be formally verified by a Hoare logic based prover.
- Possible implementers of the system. The combination of a programming language and natural language helps developers, who will use the same programming language for implementation, understand the requirements. The contracts state the semantics.

Previous publications [42,47] explain the reasons for choosing this mixed notation: unity of software construction and verification, unity of functional requirements and code, use of complementary notations geared towards different stakeholders.

Additional properties are specific to control systems:

- Specification of temporal assumptions and requirements.
- Specification of timing assumptions and requirements.
- Reuse of assumptions and requirements in stating new ones.

The basic notation is Eiffel. All the examples have been processed by the EiffelStudio IDE [1], compiled, and processed by the AutoProof verification environment. The interest of compilation is not in the generated code, since at this stage the Eiffel texts represent requirements only, but in the many consistency controls, such as type checking, of a modern compiler.

The requirements can and do take advantage of object-oriented mechanisms such as classes, inheritance and genericity.

There is sometimes an instinctive resistance to using a programming language for requirements, out of the fear of losing the fundamental difference between the goals of the two steps: programming languages normally serve for implementation, while requirements should be descriptive. The AutoReq approach, however, uses the programming language not for implementation but for specification, restricting itself to requirements patterns discussed next. The imperative nature of these patterns does not detract from this goal; empirical evidence indeed suggests [17] that operational reasoning works well not just for programmers but for other requirements stakeholders. An added benefit is the availability of program verification tools, which the approach of this article channels towards the goal of verifying requirements.

For this verification goal, there seems to be a mismatch between the standard properties that program verification tools address and the needs of control systems. Program verification generally relies on Hoare logic properties as embodied in Eiffel's Design by Contract: properties of the program state (or, for postconditions, two states). The specification of control systems generally relies on temporal and timing requirements, involving properties of an arbitrary number of (future) states of the system. A contribution of this work is to resolve the mismatch, using the programming language to emulate temporal and timing properties, through schemes described in Section 5.

4.2. Verifying requirements

Verification of AutoReq requirements relies on AutoProof [55], the prover of contracted Eiffel programs. AutoProof is a Hoare logic [22] based verifier that follows *semantic collaboration* [51] – a specification and verification methodology adapting Hoare logic to specific needs of object-oriented programming. The verification unit of AutoProof is feature with contracts. AutoReq assumptions and requirements take the form of such features, with natural language comments for better readability, to enable their direct verification with AutoProof.

Contracts for verification with AutoProof may be modular – visible to the feature's callers, and non-modular – visible only in the feature's implementation. Modular contracts take the following forms:

- *Precondition* imposes obligations on the feature's callers and benefits the callees' implementation.
- *Postcondition* guarantees benefits to the callers and imposes

obligations on the callees' implementation.

Non-modular contracts take the following forms, going back at least as far as ESC-Java [15]:

- **assume** X **end** *allows* the verification to take advantage, at the given program point, of property X, adding X to the set of properties that the prover *may* use (assumption).
- **assert** X **end** *requires* the verification to establish X before going beyond the program point, adding X to the set of properties that the prover must prove (proof obligation).

Both *precondition* and *assume* contracts add information to verifying the *postcondition* and *assert* contracts, but preconditions impose verification obligations on their own: they have to hold whenever the respective features are called. AutoReq requirements take the form of features with non-modular contracts because of their fundamental connection with the core requirements engineering terminology, as discussed further. From the purely technological perspective, AutoReq depends on the ability of AutoProof to inline callees' non-modular contracts into the callers' code.

As noted in the introduction, many software errors are requirements errors. To avoid inconsistencies, AutoReq specifications include formal properties which can be submitted to proof tools for verification. Jackson & Zave's seminal work ([27], also van Lamsweerde [33]), introduced a fundamental division of these properties:

- **Environment** (or **domain**) assumptions characterize the context in which the system must operate. The development team has no influence on them.
- **Machine** (or **system**) properties characterize what the system must do. It is the job of the development team to work on them.

Although each of these two distinctions is well-known and widely used in the corresponding sub-community of software engineering, respectively requirements and formal verification, the existing literature does not, to our knowledge, connect them. The present work, covering both requirements and verification concepts, unifies them into a single distinction:

- **assume** E **end** specifies an *environment assumption* E.
- **assert** E **end** specifies a *machine property* E.

Verifying requirements in AutoReq simply means proving that all ***assert*** hold, being permitted to take ***assume*** for granted.

Notational convention: the above notations are for presentation. The actual texts verified through the process reported in the next sections use the following standard Eiffel equivalents:

- For ***assert*** *X* ***end***, the notation in the actual Eiffel texts is ***check*** *X* ***end*** (***check*** is a standard part of Eiffel's Design by Contract mechanism).
- For ***assume*** *X* ***end***, the Eiffel notation is ***check*** *assume: X* ***end***. The *assume* tag is a standard part of the notation for programs to be verified by AutoProof. ***old*** *e*, in a routine body, denotes the value of an expression *e* on routine entry.

The only difference with verifying programs comes from the elements that appear between these assertions: in program verification, they may include any instructions; in requirements verification, we only permit patterns discussed below Section 5.1. In addition, specifications include timing properties, using the translation into classic assertions described in Sections 5.2 and 5.3.

Formal methods and notations are essential for one of the goals of this work (precision/completeness, see Section 1), but non-technical stakeholders sometimes find them cryptic at first sight, hampering other goals such as readability and ease of use. The *multirequirements* approach [42], which this article extends, addresses the problem by using complementary views, kept consistent, in various notations: formal (such as Eiffel or a specification language), graphical (such as UML) and textual (such as English). In line with this general idea, AutoReq specifications rely on systematic commenting conventions (somewhat in the style of Knuth's *literate programming* [30]). A typical example from the specification in the next section is

```
-- Assume the system
run_in_normal_mode
```

The second line is formal; the comment in the first line puts it in context. Such seemingly informal comments follow precise rules. For non-expert users, and for the present discussion, it is enough to treat them as natural-language explanations.

5. Structuring a control system specification

The mechanisms of the preceding section enable us to write the requirements for control systems and verify them. Such specifications will follow standard patterns:

- Overall structure of programs that model control systems (Section 5.1).
- Translation rules for temporal properties (Section 5.2).
- Translation rules for timing properties (Section 5.3).
- Translation rules for ASM properties (Section 5.4).

These schemes and translation patterns are fundamental to the methodology because they govern the use of the programming language. While the methodology relies on a programming language for expressing requirements, it does not use its full power, since some of its mechanisms are only relevant for programs. Programming language texts expressing requirements stick to the language subset relevant to this goal.

The translation schemes of Sections 5.2–5.4 guarantee that their output will conform to these patterns. A goal for future work (Section 8) is to formalize the input languages, timed temporal logic and ASM, and turn the translation patterns into formal rules and automatic translation tools.

Pending such formalization, we did not for now address the soundness of the translation.

5.1. Representing control systems

A control system is typically (unlike most sequential programs) repeating and non-terminating. AutoReq correspondingly uses programs of the form ***from until False loop*** *main* ***end***. The task of the requirements is then to specify *main*.

The translation uses four patterns that look like Eiffel features with non-modular (*assume* and *assert*) contracts. These patterns are not part of AutoProof, but they serve as blueprints for features that AutoProof can verify. *P1* and *P2* (Section 5.2) are time-independent (although *temporal* in the sense of temporal logic). *P3* and *P4* (Section 5.3) take timing into account. These cases suffice for the examples addressed with AutoReq so far. Translation schemes are possible for more general LTL/CTL/TPTL schemes if the need arises in the future.

The patterns use the Jackson-Zave distinction (Section 4.2) between describing an environment assumption and prescribing an expected system (machine) property. Specifically: *P1* and *P3* correspond to environment assumptions (respectively time-independent and timed); *P2* and *P4* correspond to system obligations (with the same distinction). The Eiffel translations accordingly use ***assume*** for *P1* and *P3* and ***assert*** for *P2* and *P4*. When asked to verify an AutoReq requirement, AutoProof tries to infer the *assert* statements by simulating an execution of the requirement's body to a state satisfying the *assume* statements. Fig. 2 maps the patterns according to the taxonomy of system properties used in the present article.

	Temporal Properties	Timing Properties
Environment Assumptions	P1	P3
System Obligations	P2	P4

Fig. 2. The map of AutoReq translation patterns.

5.2. Translating temporal properties

In the control systems world, the starting point for requirements is often a description expressed in a temporal logic, usually LTL [50], CTL [9], or a timed variant such as propositional temporal logic (TPTL [4]). Even if not using a specific formalism, they often state temporal properties such as *all future system states must satisfy a given condition* or *some future state must satisfy a given condition*. The LGS properties given in Section 3 are an example.

- **P1** (environment assumption)
  Consider the system running in mode *cs* under assumption *c*. The LTL formulation is $\Box(c \wedge cs)$.
- **P2** (system obligation)
  The system running in mode *cs* should immediately meet property *p*. The LTL formulation is $\Box(cs \Rightarrow \circ p)$. This property constrains the system to maintain response *p* whenever stimulus *cs* holds.

The translation scheme for *P1* is:

```
-- Assume the system
run_under_condition_c
  do
    assume
      c
    end
    main_under_conditions_cs
  end
```

where *main_under_conditions_cs* is of the form *P1* or *P3*. The *run_under_conditions* routine should be used instead of the original *main* in all requirements that talk about the system operating in mode *c*. This pattern may be useful for encoding $\Box c$ in properties of the form $\Box(\Box c \Rightarrow \diamond d)$.

The translation scheme for *P2* is:

```
-- Require the system to
immediately_meet_property_p
  do
    main_under_conditions_cs
    assert
      p
    end
  end
```

where *main_under_conditions_cs* is of the form *P1* or *P3*.

5.3. Translating timing properties

Although not all approaches to requirements take time into account, timing requirements, such as *the response time must not exceed 1 second*, are essential to the proper specification and implementation of control systems. AutoReq recognizes the following timing-related patterns:

- **P3** (environment assumption)
  Assume the system running in mode *cs* spends *t* time units to meet property *p*. The TPTL formulation is $\Box x.\ ((cs \wedge \neg\, p) \Rightarrow \circ y.\ (p \Rightarrow y = x + t))$. *x*. and *y*. record the current time of corresponding states [4].
- **P4** (system obligation)
  The system running in mode *cs* should spend no more than *t* time units to meet property *p*. In TPTL: $\Box x.\ (\Box cs \Rightarrow \diamond y.\ (p \wedge y \leq x + t))$.

The translation scheme for *P3* is:

```
-- Assume it takes t time units to take the system
from_not_p_to_p:
  do
    main_under_conditions_cs
    if (not old p and p) then
      duration := duration + t
    end
  end
```

The technique for timing system obligations of the *P4* form differs from the others by using loops as the core mechanism:

```
-- Require that
meeting_p_under_persistent_conditions_cs
-- never takes more than t time units:
  do
    from
      main_under_conditions_cs
    until
      p or (duration − old duration) > t
    loop
      main_under_conditions_cs
    end
    assert
      p and (duration − old duration) ≤ t
    end
  end
```

where *main_under_conditions_cs* is of the form *P1* or *P3*. The *(duration – **old** duration)* $> t$ exit timeout condition ensures termination of the loop, and assertion *(duration – **old** duration)* $\leq t$ checks that the timeout condition has not been reached.

The technique for handling the timing-related patterns relies on an integer, non-decreasing auxiliary variable *duration*. It has the same role as *x* and *y* in the TPTL formulations. The *duration* variable is part of the AutoReq approach – not a predefined variable nor part of AutoProof. It does not play a role in the actual execution of the system but caters to static reasoning about the system's timing properties. The *from_not_p_to_p* routine updates the value of *duration* instead of using ***assume***, which would lead to a contradiction: the prover would detect that the variable was not, in fact, updated, and would infer ***False*** from assuming the opposite.

5.4. Translating ASM properties

Abstract State Machines [13,14,21] are a commonly used specification formalism for control systems, and the treatment of the LGS case study in [6] served as a starting point for this article's own treatment of the example. The present work does not formally prove soundness of the ASM-to-Eiffel translation. The decision to work with the ASM treatment was motivated by the general ASM specifications' executability: fundamentally, they are verifiable abstractions of infinitely running control software. Such software may be implemented in a general-purpose programming language, and the present article demonstrates that such a language may serve as a verifiable abstraction of itself, in the presence of a program prover.

Below comes the ASM-to-Eiffel translation scheme. The translation scheme omits the nondeterministic version of the ASM formalism. The

original work [21] presents *"Nondeterministic Sequential Algebras"* as an extension to the basic model. As Section 1 explains, the ASM formalism serves as an implementation language example in the present discussion of AutoReq, with no intent of covering every aspect of ASMs. Nondeterministic updates seem to be inappropriate for implementing mission- and life-critical systems, such as the LGS, and control systems in general. Every possible environment's state should be predictably handled in such systems. The ASM treatment of the LGS, for example, does not use nondeterminism.

A basic ASM specification is a collection of rules taking one of three forms [20]: assignment, do-in-parallel and conditional. An ASM *assignment* reads:

$$f(t_1, ..,t_j) := t_0 \tag{1}$$

The semantics is: update the current content of location $\lambda = (f, (a_1, ..,a_j))$, where $a_{i:\ \{1..j\}}$ are values referenced by $t_{i:\ \{1..j\}}$, with the value referenced by t_0.

The Eiffel representation for an ASM location is an attribute (field) of the class; the representation for a location update is an attribute assignment.

The ASM *do-in-parallel* operator applies several assignments in one step. Eiffel offers no native support for do-in-parallel, but it can emulate one sequentially without changing the behavior. The following example gives intuition behind the translation idea:

$$a, b := max(a - b, b), min(a - b, b) \tag{2}$$

The instruction in Eq. (2), when run infinitely, reaches the fixpoint in which a contains the greatest common divisor of a and b. The Eiffel translation of this instruction is:

```
local
  a_intermediate, b_intermediate: INTEGER
do
  a_intermediate := max (a−b, b)
  b_intermediate := max (a−b, b)
  a := a_intermediate
  b := b_intermediate
end
```

The generalization should be clear at this point: instead of updating directly the locations, introduce and update intermediate local variables, and then assign them to the locations.

The translation of an ASM *conditional* (***if*** *t* ***then*** *R* ***else*** *Q*) is an Eiffel conditional instruction.

The ASM-to-Eiffel translation scheme scales out to the multiple classes case. The translation overhead in this case consists of implementing assigner procedures for the supplier classes' attributes. The assigner procedures will make it possible for the clients to update the suppliers' attributes while keeping them consistent. The LGS example is simple enough to avoid the multiple classes case, which is why the present work never applies this translation rule.

6. The landing gear system in AutoReq

Equipped with the AutoReq mechanisms as described, we can now see the core elements of the AutoReq specification of the LGS example. The entire example is available in a public GitHub repository [45].

6.1. Normal mode of execution

Execution runs in *normal mode* if all the parameter values are in the expected ranges and meet the system invariant. Application of the *run_under_condition_c* pattern results in the following Eiffel model of normal mode:

```
-- Assume the system
run_in_normal_mode
  do
    -- the handle status range:
    assume
      handle_status = up_position or
      handle_status = down_position
  end
  -- the door status range:
  assume
    door_status = closed_position or
    door_status = opening_state or
    door_status = open_position or
    door_status = closing_state
  end
  -- the gear status range:
  assume
    gear_status = extended_position or
    gear_status = extending_state or
    gear_status = retracted_position or
    gear_status = retracting_state
  end
  -- the gear may extend or retract only with the door open:
  assume
    (gear_status = extending_state or gear_status = retracting_state)
      implies door_status = open_position
  end
  -- closed door assumes retracted or extended gear
  assume
    door_status = closed_position implies
      (gear_status = extended_position or gear_status = retracted_position)
  end
  main
end
```

The first three *assume* express that attribute values fall into specific ranges. The last two express the LGS invariant. Ranges, the invariant and the definition of normal mode come from the original. *run_in_normal_mode* is a multiple application of the *run_under_condition_c* pattern (Section 5.2). It wraps around *main* to make additional assumptions before calling it.

6.2. Timing properties

The ASM treatment of the LGS case study ignores timing properties stated in the original description. For a practical system, timing is essential; an otherwise impeccable LGS that takes two hours to perform *extend landing gear* would not be attractive. We rely on AutoReq's timing mechanisms of the AutoReq methodology (Section 5.3) and the *from_not_p_to_p* pattern (Section 5.3). Timing values, e.g. 8 units for door closing, are for illustration only. Each of the translations that follow are produced by applying the same pattern, which is why only the first translation is accompanied by a detailed explanation.

- *It takes 8 time units for the door to close.* Replacing *p* with *door_status = closed_position*, and *t* with *8* in the *from_not_p_to_p* pattern yields:

```
-- Assume it takes 8 time units to take the door
from_open_to_closed -- position:
  do
    run_in_normal_mode
    if (old door_status ≠ closed_position and
       door_status = closed_position) then
       duration := duration + 8
    end
  end
```

- *It takes 12 time units for the door to open*:

```
--Assume it takes 12 time units to take the door
from_closed_to_open -- position:
  do
    from_open_to_closed
    if (old door_status ≠ open_position and
       door_status = open_position) then
      duration := duration + 12
    end
  end
```

- *It takes 10 time units for the gear to retract*:

```
--Assume it takes 10 time units to take the gear
from_extended_to_retracted -- position:
  do
    from_closed_to_open
    if (old gear_status ≠ retracted_position and
       gear_status = retracted_position) then
      duration := duration + 10
    end
  end
```

- *It takes 5 time units for the gear to extend*:

```
-- Assume it takes 5 time units to take the gear
from_retracted_to_extended -- position:
  do
    from_extended_to_retracted
    if (old gear_status ≠ extended_position and
     gear_status = extended_position) then
    duration := duration + 5
    end
  end
```

from_retracted_to_extended will include all the previously stated *assume* instructions together with *main*.

6.3. Baseline requirements

Section 3 introduced a set of core LGS requirements, $R_{11}bis$ to R_{22}, which we now express in AutoReq. $R_{11}bis$ and R_{21} talk about the system running with the handle pushed down. Application of the *run_under_condition_c* pattern (Section 5.2) with *handle_status = down_position* for *c* results in the following routine to model the required mode of operation:

```
-- Assume the system
run_with_handle_down
  do
    assume handle_status = down_position end
    from_retracted_to_extended
  end
```

run_with_handle_down is an application of the *run_under_condition_c* pattern (Section 5.2). It calls *from_retracted_to_extended* to include all assumptions so far.

Now that the execution mode with the handle pushed down is formally defined, it is possible to express the requirements in terms of it. Property R_{21} requires the controller to prevent retraction immediately whenever the handle is pushed down. Application of the *immediately_meet_property_p* pattern (Section 5.2) with *gear_status /= retracting_state* for *p* yields, for R_{21}:

```
-- Require the system to
never_retract_with_handle_down
  do
    run_with_handle_down
    assert gear_status ≠ retracting_state end
    -- known as R_{21}
  end
```

$R_{11}bis$ requires the system eventually to extend the gear and close the door if the handle stays down. The absence of timing makes it unsuitable for the specification of control systems: we need to specify an upper bound on the time the system may spend on gear extension. That bound is the sum of the maximal times for door closing, door opening and gear extension. Under earlier assumptions, this value is 25. Applying *meeting_p_under_persistent_conditions_cs* (Section 6.2) with *gear_status = extended_position* and *door_status = closed_position* for *p*, *run_with_handle_down* for *main_under_conditions_cs* and *25* for *t* turns $R_{11}bis$ into:

Requirements $R_{12}bis$ and R_{22} talk about the system running with the handle pulled up. Application of *run_under_condition_c* (Section 5.2) with

```
-- Require that
extension_duration
-- never takes more than 25 time units:
 do
  from
   run_with_handle_down
  until
   (gear_status = extended_position and door_status = closed_position) or
   (duration − old duration) > 25
  loop
   run_with_handle_down
  end
  assert gear_status = extended_position end
  assert door_status = closed_position end
  assert (duration − old duration) ≤ 25 end
  -- known as R_{11}bis
 end
```

handle_status = *up_position* for *c* yields:

```
-- Assume the system
run_with_handle_up
 do
  assume
   handle_status = up_position
  end
  from_retracted_to_extended
 end
```

The rest of the requirements can rely on the specification of the execution mode with handle up, as we have now obtained.

R_{22} requires the system to prevent immediate extension whenever the handle is pulled up. Application of *immediately_meet_property_p* (Section 5.2) with *gear_status* /= *extending_state* for *p* yields, for R_{22}:

```
-- Require the system to
never_extend_with_handle_up
 do
  run_with_handle_up
 assert
  gear_status ≠ extending_state
 end
  -- known as R_{22}
 end
```

$R_{12}bis$ requires the system eventually to retract the gear and close the door if the handle stays up. Like $R_{11}bis$, it does not include timing. The upper bound for $R_{12}bis$ is the sum of the maximal times for door closing, door opening and gear extension, 30 from earlier assumptions. Applying *meeting_p_under_persistent_conditions_cs* (Section 6.2) with *gear_status* = *retracted_position and door_status* = *closed_position* for *p*, with *run_with_handle_up* for *main_under_conditions_cs* and *30* for *t* yields:

```
-- Require that
retraction_duration
-- never takes more than 30 time units:
 do
  from
   run_with_handle_up
  until
   (gear_status = retracted_position and door_status = closed_position) or
   (duration − old duration) > 30
  loop
   run_with_handle_up
  end
  assert
   gear_status = retracted_position and
   door_status = closed_position and
   (duration − old duration) ≤ 30
   -- known as R_{12}bis
  end
 end
```

6.4. Complementary requirements

$R_{11}bis$ and $R_{12}bis$ talk about reaching a desired state under some conditions, but not about preserving it. For example, even if the gear becomes extended and the door closed with the handle down, this situation must not change without the handle pulled up. The following application of *immediately_meet_property_p* (Section 5.2) with *gear_status = extended_position and door_status = closed_position* for *p* captures this property:

```
-- Require the system to
keep-gear-extended-door_closed-with_handle-down
  do
    run_with_handle_down_gear_extended_door_closed
    assert
      gear_status = extended_position and
      door_status = closed_position
    end
  end
```

under the assumption that the doors are already closed, the gear is extended, and the handle is down. Application of *run_under_condition_c* (Section 5.2) with *gear_status = extended_position and door_status = closed_position* for *c* yields, for this assumption:

```
-- Assume the system
run_with_handle_down_gear_extended_door_closed
  do
    assume
      gear_status = extended_position and
      door_status = closed_position
    end
    run_with_handle_down
  end
```

The state with the gear retracted, the door closed and the handle pulled up should be stable without pushing the handle down. The following application of *immediately_meet_property_p* (Section 5.2) with *gear_status = retracted_position and door_status = closed_position* for *p* yields:

```
-- Require the system to
keep_gear_retracted_door_closed_with_handle_up
  do
    run_with_handle_up_gear_retracted_door_closed
    assert
      gear_status = retracted_position and
      door_status = closed_position
    end
  end
```

under the assumption that the doors are already closed, the gear is retracted, and the handle is up. Application of *run_under_condition_c* pattern (Section 5.2) with *gear_status = retracted_position and door_status = closed_position* for *c* yields, for this assumption:

```
-- Assume the system
run_with_handle_up_gear_retracted_door_closed
do
  assume
    gear_status = retracted_position and
    door_status = closed_position
  end
  run_with_handle_up
end
```

6.5. An error in the ground model

Contracts do not just yield expressive power: they also make automatic verification possible in the AutoReq approach thanks to AutoProof. One of the principal potential benefits would be to uncover errors in the requirements.

Our work on the LGS example shows that this benefit is not just a theoretical possibility. Applying the AutoReq method and tools to the published ASM specification of the LGS system [6] uncovered an error. The verification process applied the following sequence of steps.

Start from the ASM specification. The language in which the ASM specification is expressed contains syntactic sugar in addition to the standard ASM operators. The first step consisted of analyzing these additional constructs to understand how they should translate to Eiffel.

Translate it into Eiffel. This step consisted of manual translation of the specification and the requirements to Eiffel. One can find the original ASM specification in an online archive [39], inside the *LandingGearSystemGround.asm* file. File *ground_model.e* in the GitHub repository [45] contains the result of the translation.

Verify it with AutoProof. Note that AutoProof, by default, performs modular contract-based verification. AutoReq specification techniques rely on *assume* and *assert* rather than traditional contracts. These specification techniques require tuning AutoProof command-line options. The GitHub repository [45] with the Eiffel translation includes a *readme* file that says in detail how to launch AutoProof.

Identify the error. When AutoProof reports a verification failure, it does not point at its root cause. The last step was devoted to identifying that cause.

The error uncovered by this procedure is subtle and revealing: *The specification does not meet the $R_{11}bis$ requirement, which states that pushing the handle down should lead to the gear extended and the door closed.* Normally, when the crew pushes the LGS handle down, the controller should initiate the gear extension process. Regardless of the initial system's state, this process should end up correctly – so that in the end the gear is extended and the LGS latch is closed.

There exists, however, a state from which the erroneous ASM specification will not bring the system to the correct configuration. This state corresponds to a situation in which the gear has just been retracted, the door is closing, and the crew decides to cancel retraction by pushing the handle down. A correctly working system would cancel the retraction sequence and initiate gear extension. State 15 on Fig. 3 illustrates this situation: the *start_opening* outgoing action cancels the door closing process initiated by action *start_closing* back in state 7. The state machine proceeds with the gear extension procedure. The erroneous ASM specification models a system that waits for the crew to pull the handle up again to let the system complete the gear retraction process. State 15 on Fig. 4 features only one outgoing transition: pulling the handle up again. Instead of canceling the door closing process (Fig. 3), the system starts waiting for the crew to pull the handle up. Imagine a situation in which the crew tries to retract the gear during take-off, and some physical obstacle prevents the latch from closing completely. In this case a possible solution might be to extend the gear back, and then try to retract it again. A real controller implemented around the erroneous specification would make extension with the latch partially closed impossible.

The published Eiffel translation of the specification does not have the error. To catch it with the AutoReq method one needs first to introduce the error back by commenting out two lines in the *open_door* routine of the Eiffel translation:

```
when closing_state then
door_status := opening_state
```

and then submit routine *extension_duration* to the AutoProof tool; the verification will fail. The "README" file in the accompanying GitHub repository [45] provides detailed instructions on submitting AutoReq requirements to AutoProof. Internally, AutoProof transforms the Eiffel routine to Boogie code and submits it to the Boogie executable [8]. The Boogie executable converts its input to first-order logic formulae and

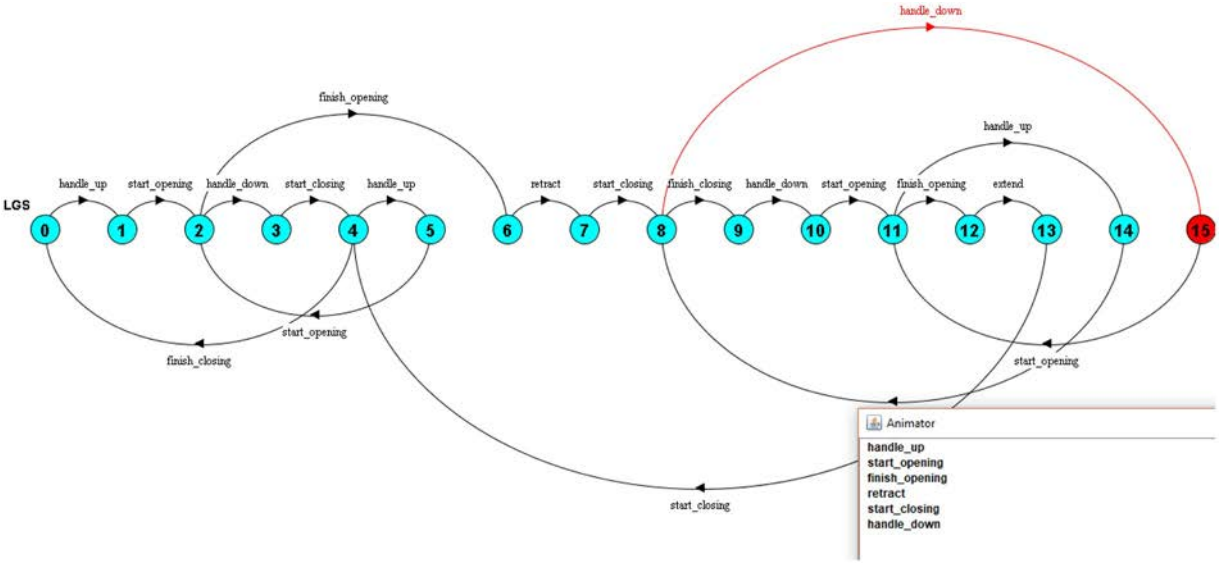

Fig. 3. A correctly working LGS state machine. Pushing the handle down cancels the gear retraction process and initiates gear extension. The bottom-right box contains the trace leading to state 15.

submits them to the Z3 SMT solver [44].

AutoProof detects the error in the following major steps:

Inline the unqualified calls inside of the *extension_duration* routine to the level of attribute updates and ***assume*** statements.
Unroll the loop inside of *extension_duration*. How much to unroll is a configurable setting; the default configuration suffices for the LGS example.
Check the ***assert*** statements based on the outcome from the *Inline* step.

The intent of applying AutoReq to this example was not to look for errors but to try out the approach, illustrate it on a widely used problem, and compare it with other treatments of that problem. No error had been reported and we did not expect to find one. To ascertain its presence, we contacted one of the authors of the original article describing the ASM implementation. He confirmed the presence of the error in the paper. (He also noted that the private repository used by his colleagues and him had a correct specification.)

7. Related work

7.1. Similar studies

The ASM treatment of the LGS example comes from a collection including other treatments [12], such as Event-B [31,36,53], Fiacre [11] and Hybrid Event-B [7]. The original collection [12] discusses pros and cons of these approaches, and the present article does not repeat that discussion. AutoReq complements these approaches with the following:

- Language reuse: AutoReq captures temporal and timing properties in a general purpose programming language. This will inevitably save resources for software teams that want to apply formal methods.
- Technology reuse: AutoReq relies on AutoProof, a Hoare logic based program prover. The original use case of AutoProof was specifying and verifying programs according to the principles of Design by Contract. With AutoReq, software teams can use the tool throughout

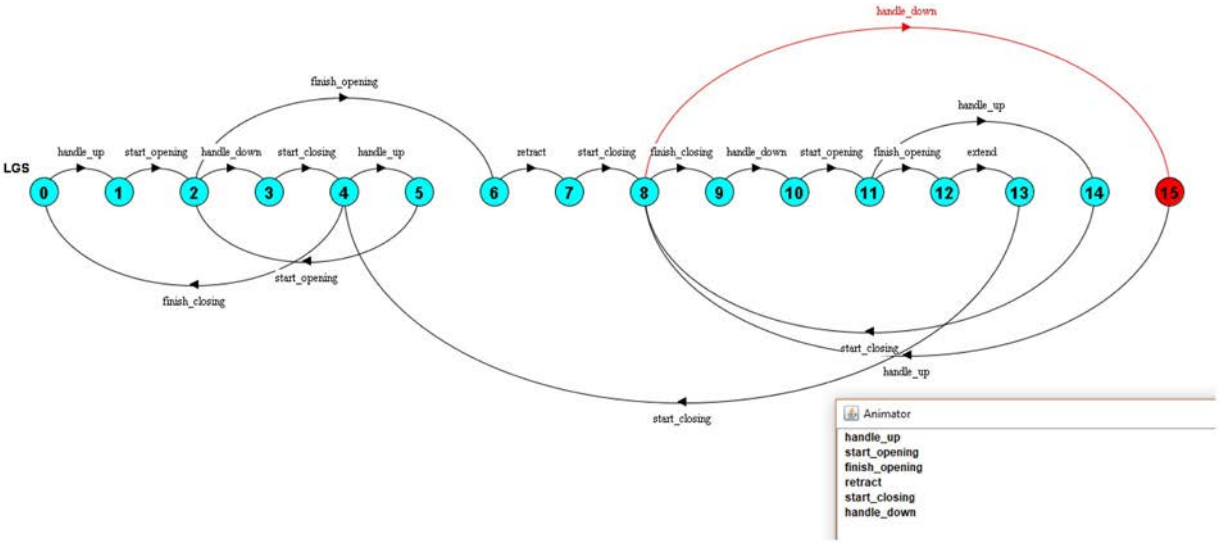

Fig. 4. The erroneous LGS state machine. Pushing the handle down fails to cancel the gear retraction process. It puts the system to waiting for the crew to pull the handle up again. The bottom-right box contains the trace leading to state 15.

the whole software lifecycle, starting from the requirements phase.

- Specification reuse: AutoReq makes it possible to avoid copying-and-pasting already stated assertions through the standard routine call mechanism, familiar to any post-Assembly programmer.
- Implementation reuse: AutoReq does not require translating programs to models and back for further formal verification. If a change in the program breaks an AutoReq requirement, the prover will immediately notice this.

These advantages need stronger support in the form of successful industrial applications of AutoReq. Such applications may also uncover additional problems to solve. The application of AutoReq to the LGS example discussed in the present article inherits the questionable assumptions (Section 3) from the original work by Arcaini et al. Applying AutoReq to an example with weaker assumptions would provide more evidence of its benefits.

The applicability studies will follow the present article that focuses on illustrating the approach alone. Combining the first description of AutoReq with its applicability studies would bear the risk of making the article difficult to read.

7.2. Existing formalisms

Reasoning about programs, imperative and concurrent, has been the focus of computer science researchers for decades [28], and it traces back as early as Turing's work [29]. Different techniques have been developed over time, and it soon became clear that, while *post facto* verification can be successful for small programs, an effective verification strategy should support and be part of the software development itself and be fully embedded in the process.

The AutoReq method follows this idea and relies on DbC verification; however, one should understand that DbC is not well suited for control systems as it is. The possibility of unexpected changes in the values of environment-controlled variables introduces the gap between DbC and control systems. Traditional DbC relies on invariant-based reasoning, on the principle of invariant stability [51]: it should be impossible for an operation to make an object inconsistent without modifying the object. This principle does not work with control systems because of the unpredictable environment-controlled variables. In other words, any attempt to constrain an environment-controlled variable through a contract will inevitably lead to the contract's failure.

Control systems communicate asynchronously with the environment. This introduces another gap with DbC, which is designed from the beginning to deal with synchronous software. For non-life-critical systems [27] one may sacrifice the asynchrony under additional assumptions [48], but the Landing Gear System does not fall into this category.

An interesting technique for including environment properties is the notion of monitor introduced by Zave [58]. A monitor is an executable requirement that runs in a dedicated process and observes the system from outside logging possible anomalies. A monitor continuously polls the state of nondeterministic variables and checks if the system evolves accordingly. This is, however, a run-time mechanism; in the present work, we seek requirements techniques that lend themselves to static verification.

The general aspiration towards sound static verification resulted in numerous modeling approaches that rely on a declarative logic. Alloy [25] is one of these declarative modeling languages, based on first-order logic, that are used to express complex behavior of software systems. Alloy is a successor of Z [2] with its own formal syntax and semantics, that adds automatic verification and tool support to Z specifications. A model created in Alloy can indeed be automatically checked for correctness by using a dedicated tool: the *Alloy Analyzer*, a SAT-based constraint solver that provides fully automatic simulation and checking. Alloy is one of the tools used for *requirements verification*. There are several examples of successful applications of the modeling languages in different fields: from pedagogical to enterprise modeling to transportation. A list documenting some of these applications can be found in [26].

The declarative view simplifies static reasoning, but the system will eventually have to physically operate. C. A. R. Hoare introduced an imperative logic to statically reason about software way back in 1969. This invention has been treated as a verification mechanism. We are interested in requirements specification notations. The recent notion of seamless requirements [47] proposes a use of generalized Hoare triples called specification drivers [46] as a requirements notation.

The AutoReq method steps forward by applying the idea of seamless requirements to the nondeterministic setting. It empowers the operational view of Pamela Zave on requirements with AutoProof – a Hoare logic based prover of Eiffel programs with contracts that relies on the Boogie technology [34]. In AutoReq a requirement is a routine enriched with ***assume*** statements capturing environment assumptions and ***assert*** statements that capture the obligations for AutoProof corresponding to the assumptions. The resulting method respects environment-controlled phenomena as monitors do but does not assume the requirements to physically run. The AutoReq method will benefit the development process even when there is no static prover like AutoProof: an operational requirement will become a subject to testing as a parameterized unit test (PUT) [54]. The testing will consist in this case of running the requirement in the simulated environment described in its ***assume*** statements.

7.3. Timing properties

Modeling real-time computation and related requirements has been a well-investigated matter for long [57]. Representation of real-time requirements, expressed in general or specific form, is a challenging task that has been attacked through several formalisms both in sequential and concurrent settings, and in a broad set of application domains. The difficulty (or impossibility) to fully represent general real-time requirements other than in natural language or making use of excessively complicated formalisms (unsuitable for software developers), has been recognized.

In [38] the domain of real-time reconfiguration of systems is discussed, emphasizing the necessity of adequate formalisms. The problem of modeling real time in the context of services orchestration in Business Process, and in presence of abnormal behavior has been examined in [37] and [18] by means, respectively, of process algebra and temporal logic. Modeling protocols also requires real-time aspects to be represented [10]. Event-B has also been used as a vector for real-time extension [24] to handle control systems requirements.

In all these studies, the necessity emerged of focusing on specific typology of requirements using ad-hoc formalisms and techniques and making use of abstractions. The notion of *real-time* is often abstracted as *number of steps*, a metric commonly used.

The AutoReq method works with the explicit notion of time distance between events by stating operational assumptions on the environment; it also supports the abstraction of time as number of steps through finite loops with integer counters.

8. Conclusions and future work

The approach presented above is a comprehensive method for requirements analysis based on ideas from modern object-oriented software engineering and the application of a seamless software process that relies on the notation of a programming language as a modeling tool throughout the software process. The work also introduces the notion of verifying requirements and shows how to use a program prover to perform the verification. In addition, it connects fundamental concepts, heretofore considered independent, from two different areas of research: verification (*assume/assert*) and requirements (*environment/machine*).

The work is subject to the following limitations, also suggesting areas of improvement:

- While the idea of seamless requirements has been widely applied, its AutoReq development as described here needs more validation on diverse examples in an industrial setting, with actual stakeholders involved.
- The patterns given are not necessarily complete; here too experience with more examples is necessary to determine if there is a need for other patterns.
- The idea of using a programming language for requirements runs counter to accepted ideas; while there are strong arguments supporting it, and ample discussions in some of the OO literature, some people may still hesitate to adopt it.
- More work is required to determine how applicable AutoReq would be to a software process relying on technologies other than Eiffel and AutoProof. In line with this goal, we applied AutoReq [19] to the London Ambulance System case [3,35] and continue working on other examples.
- As discussed in Section 5, parts of the process may benefit from more automation. Such further tool support is currently under development.

With these reservations, we believe that the article and the case study demonstrate the benefits and contributions listed in the introduction and point to a promising approach to producing and verifying effective requirements for control systems.

Acknowledgment

We are indebted to the authors of the ASM version of the LGS case study [6] for their careful work on this problem. We are particularly grateful to Professor Angelo Gargantini for his openness, patience and insights in discussing the ASM work with us.

The authors are thankful to the administrations of Innopolis University and Toulouse University for the funding that made this work possible.

References

[1] https://www.eiffel.org/.
[2] J. Abrial, S.A. Schuman, B. Meyer, Specification language, On the Construction of Programs, (1980), pp. 343–410.
[3] D. Alrajeh, J. Kramer, A. Russo, S. Uchitel, Elaborating requirements using model checking and inductive learning, IEEE Trans. Software Eng. 39 (3) (2013) 361–383.
[4] R. Alur, T.A. Henzinger, A really temporal logic, J. ACM 41 (1) (1994) 181–204.
[5] Y.A. Ameur, K. Schewe (Eds.), Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings, vol. 8477, Lecture Notes in Computer Science, Springer, 2014doi:10.1007/978-3-662-43652-3.
[6] P. Arcaini, A. Gargantini, E. Riccobene, Rigorous development process of a safety-critical system: from ASM models to Java code, STTT 19 (2) (2017) 247–269.
[7] R. Banach, The landing gear case study in hybrid Event-B, in: [5], pp. 126–141. 10.1007/978-3-662-43652-3.
[8] M. Barnett, B.E. Chang, R. DeLine, B. Jacobs, K.R.M. Leino, Boogie: a modular reusable verifier for object-oriented programs, FMCO, Lecture Notes in Computer Science 4111 Springer, 2005, pp. 364–387.
[9] M. Ben-Ari, A. Pnueli, Z. Manna, The temporal logic of branching time, Acta Inf. 20 (1983) 207–226.
[10] M. Berger, K. Honda, The two-phase commitment protocol in an extended pi-Calculus, Electr. Notes Theor. Comput. Sci. 39 (1) (2000) 21–46.
[11] B. Berthomieu, S. Dal Zilio, Ł. Fronc, Model-checking real-time properties of an aircraft landing gear system using fiacre, in: [5], pp. 110–125. doi:10.1007/978-3-662-43652-3.
[12] F. Boniol, V. Wiels, The landing gear system case study, in: [5], pp. 1–18. doi:10.1007/978-3-662-43652-3.
[13] E. Börger, A. Raschke, Modeling Companion for Software Practitioners, Springer, 2018.
[14] E. Börger, R.F. Stärk, Abstract State Machines. A Method for High-Level System Design and Analysis, Springer, 2003.
[15] D.R. Cok, J. Kiniry, ESC/Java2: uniting ESC/Java and JML, CASSIS, Lecture Notes in Computer Science 3362 Springer, 2004, pp. 108–128.
[16] P. Dhaussy, C. Teodorov, Context-aware verification of a landing gear system, in: Ameur and Schewe [5], pp. 52–65. doi:10.1007/978-3-662-43652-3.
[17] D. Fahland, D. Lübke, J. Mendling, H.A. Reijers, B. Weber, M. Weidlich, S. Zugal, Declarative versus imperative process modeling languages: the issue of understandability, BMMDS/EMMSAD, Lecture Notes in Business Information Processing 29 Springer, 2009, pp. 353–366.
[18] L. Ferrucci, M.M. Bersani, M. Mazzara, An LTL semantics of business workflows with recovery, Software Paradigm Trends (ICSOFT-PT), 2014 9th International Conference on, IEEE, 2014, pp. 29–40.
[19] F. Galinier, Specification of the London ambulance system in AutoReq, 2018, (https://gitlab.com/fgalinier/LAS).
[20] Y. Gurevich, Sequential abstract-state machines capture sequential algorithms, ACM Trans. Comput. Log. 1 (1) (2000) 77–111.
[21] Y. Gurevich, J.K. Huggins, Evolving Algebras and Partial Evaluation, IFIP Congress (1), IFIP Transactions A-51 North-Holland, 1994, pp. 587–592.
[22] C.A.R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580.
[23] I.F. Hooks, K.A. Farry, Customer-Centered Products: Creating Successful Products Through Smart Requirements Management, Amacom Books, 2001.
[24] A. Iliasov, A.B. Romanovsky, L. Laibinis, E. Troubitsyna, T. Latvala, Augmenting event-B modelling with real-time verification, FormSERA@ICSE, IEEE, 2012, pp. 51–57.
[25] D. Jackson, Software Abstractions - Logic, Language, and Analysis, MIT Press, 2006.
[26] D. Jackson, Alloy applications, 2017, (http://alloy.mit.edu/alloy/citations/case-studies.html).
[27] M. Jackson, P. Zave, Deriving specifications from requirements: an example, ICSE, ACM, 1995, pp. 15–24.
[28] C.B. Jones, The early search for tractable ways of reasoning about programs, IEEE Ann. History Comput. 25 (2) (2003) 26–49.
[29] C.B. Jones, Turings 1949 paper in context, Conference on Computability in Europe, Springer, 2017, pp. 32–41.
[30] D.E. Knuth, Literate programming, Comput. J. 27 (2) (1984) 97–111.
[31] L. Ladenberger, D. Hansen, H. Wiegard, J. Bendisposto, M. Leuschel, Validation of the ABZ landing gear system using ProB, STTT 19 (2) (2017) 187–203.
[32] M. Lake, Epic failures: 11 infamous software bugs, ComputerWorld (Sept) (2010).
[33] A. van Lamsweerde, Requirements Engineering - From System Goals to UML Models to Software Specifications, Wiley, 2009.
[34] K.R.M. Leino, This is Boogie 2, Manuscript KRML 178 (131) (2008).
[35] E. Letier, Reasoning about agents in goal-oriented requirements engineering, Ph.D. thesis, Université catholique de Louvain, 2001.
[36] A. Mammar, R. Laleau, Modeling a landing gear system in Event-B, STTT 19 (2) (2017) 167–186.
[37] M. Mazzara, Timing issues in web services composition, EPEW/WS-FM, Lecture Notes in Computer Science 3670 Springer, 2005, pp. 287–302.
[38] M. Mazzara, A. Bhattacharyya, On modelling and analysis of dynamic reconfiguration of dependable real-time systems, Dependability (DEPEND), 2010 Third International Conference on, IEEE, 2010, pp. 173–181.
[39] F. Methods, S.E. Laboratory, Landing gear system ASM specification, 2014, (http://fmse.di.unimi.it/sw/landingGearSystem.zip).
[40] B. Meyer, Applying "design by contract", IEEE Comput. 25 (10) (1992) 40–51.
[41] B. Meyer, Object-Oriented Software Construction, 2nd Edition, Prentice-Hall, 1997.
[42] B. Meyer, Multirequirements, Modelling and Quality in Requirements Engineering: Essays dedicated to Martin Glinz on the occasion of his 60th birthday, Verl.-Haus Monsenstein u. Vannerdat, 2013.
[43] F. Modugno, N.G. Leveson, J.D. Reese, K. Partridge, S.D. Sandys, Integrated safety analysis of requirements specifications, Requir. Eng. 2 (2) (1997) 65–78.
[44] L.M. de Moura, N. Bjørner, Z3: an efficient SMT solver, TACAS, Lecture Notes in Computer Science 4963 Springer, 2008, pp. 337–340.
[45] A. Naumchev, Landing gear system ground model specification and requirements in Eiffel, 2017, (https://github.com/anaumchev/lgs_ground_model).
[46] A. Naumchev, B. Meyer, Complete contracts through specification drivers, TASE, IEEE Computer Society, 2016, pp. 160–167.
[47] A. Naumchev, B. Meyer, Seamless requirements, Comput. Lang., Syst. Struct. 49 (2017) 119–132.
[48] A. Naumchev, B. Meyer, V. Rivera, Unifying requirements and code: an example, Ershov Memorial Conference, Lecture Notes in Computer Science 9609 Springer, 2015, pp. 233–244.
[49] P.G. Neumann, Computer-Related Risks, Addison-Wesley, 1995.
[50] A. Pnueli, The temporal logic of programs, FOCS, IEEE Computer Society, 1977, pp. 46–57.
[51] N. Polikarpova, J. Tschannen, C.A. Furia, B. Meyer, Flexible invariants through semantic collaboration, FM, Lecture Notes in Computer Science 8442 Springer, 2014, pp. 514–530.
[52] J.E. Rumbaugh, M.R. Blaha, W.J. Premerlani, F. Eddy, W.E. Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, 1991.
[53] W. Su, J. Abrial, Aircraft landing gear system: approaches with Event-B to the modeling of an industrial system, STTT 19 (2) (2017) 141–166.
[54] N. Tillmann, W. Schulte, Parameterized unit tests, ESEC/SIGSOFT FSE, ACM, 2005, pp. 253–262.
[55] J. Tschannen, C.A. Furia, M. Nordio, N. Polikarpova, AutoProof: auto-active functional verification of object-oriented programs, TACAS, Lecture Notes in Computer Science 9035 Springer, 2015, pp. 566–580.
[56] K. Walden, J. Nerson, Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems, Prentice-Hall, 1994.
[57] H. Yamada, Real-time computation and recursive functions not real-time computable, IRE Trans. Electron. Comput. 11 (6) (1962) 753–760.
[58] P. Zave, An operational approach to requirements specification for embedded systems, IEEE Trans. Software Eng. 8 (3) (1982) 250–269.